Lab 1 Report

Tuan Nguyen

Deep Learning Spring 2017

Dr. Hagan

### February 10, 2017

# Basic Lua

1. What does interactive interpreter mean?.

It means we can type the command and our command will be interpreted (and executed) immediately when press Enter. It likes we are talking with the compiler. It pretty much similar to Python, Matlab Command and DOS Cmd.

2. What are ”chunks” in Lua?

Chunk is a sequence of statements

3. What is a global variable?

Global variables is visible to all programs and do not need declaration. It is default unless local keyword provided.

4. Name the types that exist in Lua.

8 basic types: nil, boolean, number, string, userdata, function, thread, table.

5. What are local variables and blocks in Lua?

local variable has its scope limited to the block where it is declared. A block is the body of a control structure, function or a chunk.

6. Identify differences between the interactive interpreter and Torch debugger. Describe one

example.

Interpreter compiles and run commands and then returns results back to users when finished. Users can see results (and error messages if bugs exist) only after execution done.

Debugger is based on interpreter but helps users “watch” the state of programs after each steps. The state can be variable, register, memory and so on.

7. Where can the dofile command can be used (interactive interpreter or Torch Debugger)?

Explain the advantages of using this command

dofile can be used in interpreter to execute a Lua file. (Besides, in my opinion, when we press F6(execute file) in Torch debugger, it calls dofile). It run a whole Lua file as a Lua chunk (batch execution), not one-by-one. Therefore, it is more convenient, time-saving and used to automate the process.

# Q3. Testfile.lua

**Output**

Program starting as '"/opt/zbstudio/bin/linux/x64/lua" -e "io.stdout:setvbuf('no')" "/home/martin/Desktop/tuandn/deep\_learning/lab1/testfile.lua"'.

Program 'lua' started in '/opt/zbstudio/myprograms' (pid: 12086).

10

20

30

Program completed in 0.02 seconds (pid: 12086).

**Explain**

The function list\_iter() takes a table as its input, check the index and return a function that in tern, return a value in a table if index is valid. The main code initializes a table with 3 elemnts, then calls list\_iter() function, passes the table and then loop through the table to print its elements.

# Q4. Examine code

The code uses while loop to print numbers from 2--> 20 then print 10 and print 10. The output matches with this examination. The confusion might come from using var as both global and local. However, in while loop, when a local var is declared, it masks (or hides) the global var.

Program starting as '"/opt/zbstudio/bin/linux/x64/lua" -e "io.stdout:setvbuf('no')" "/home/martin/Desktop/tuandn/deep\_learning/lab1/q4.lua"'.

Program 'lua' started in '/opt/zbstudio/myprograms' (pid: 12131).

2

4

6

8

10

12

14

16

18

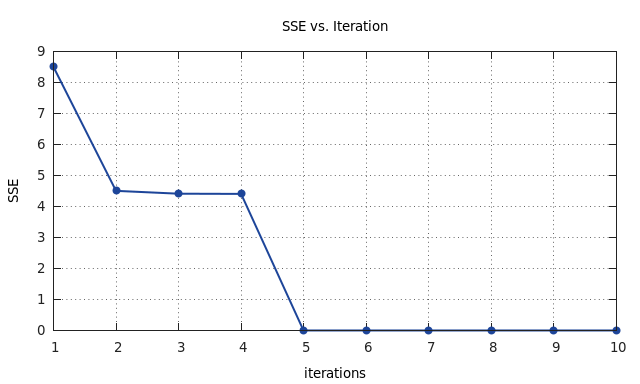
20

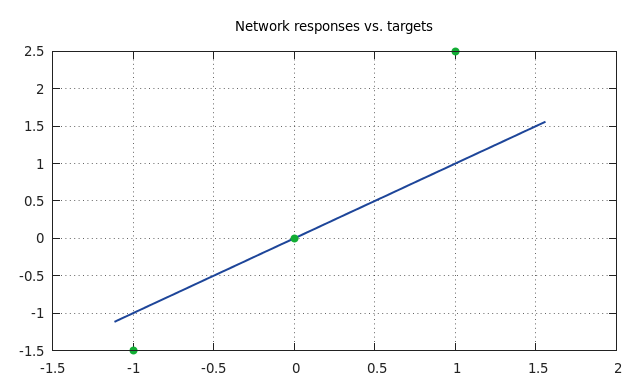
10

10

Program completed in 0.01 seconds (pid: 12131).

# Q5. Lua for HW3





# Q6. XOR functions

1. GPU test

--CUDA run for XOR\_optim

require 'cunn'

….

batchInputs = batchInputs:cuda()

batchLabels = batchLabels:cuda()

model:cuda()

criterion:cuda()

….

x = torch.CudaTensor({

{ 0.5, 0.5},

{ 0.5, -0.5},

{-0.5, 0.5},

{-0.5, -0.5}

})

print(model:forward(x))

require 'cunn'

….

mlp:cuda()

criterion:cuda()

….

input = input:cuda()

output = output:cuda()

….

x = torch.CudaTensor({

{ 0.5, 0.5},

{ 0.5, -0.5},

{-0.5, 0.5},

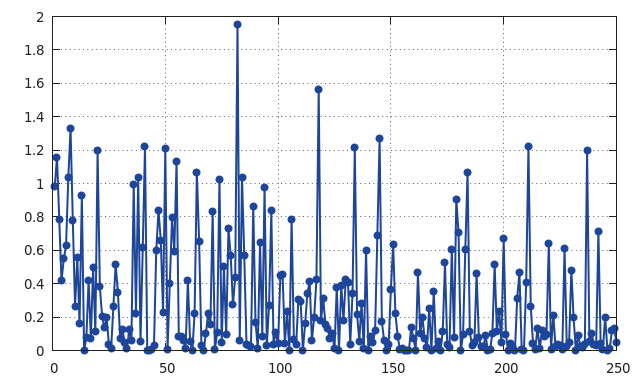
{-0.5, -0.5}

})

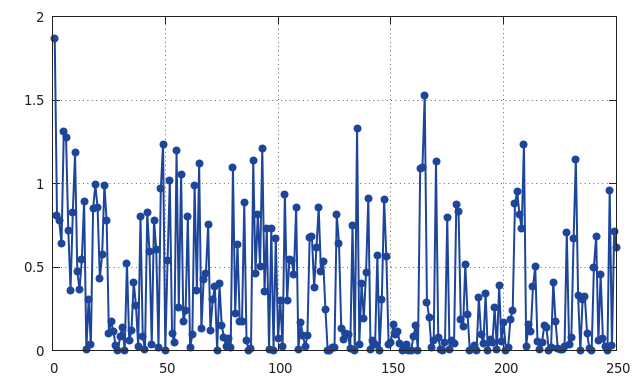
print(mlp:forward(x))

## 2. SEE vs. Iterations (XOR\_Stochastic)

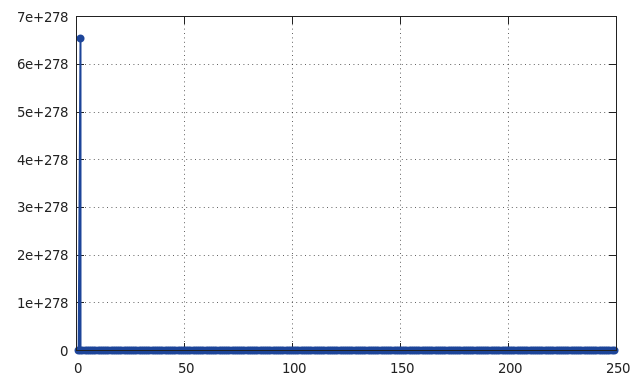
a. Learning Rate = 0.01 (default)



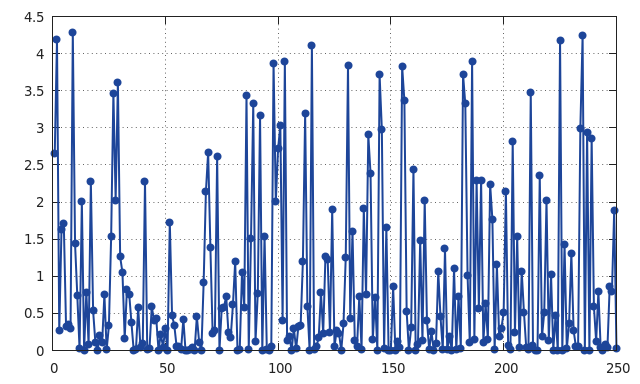
b. Learning Rate = 0.005

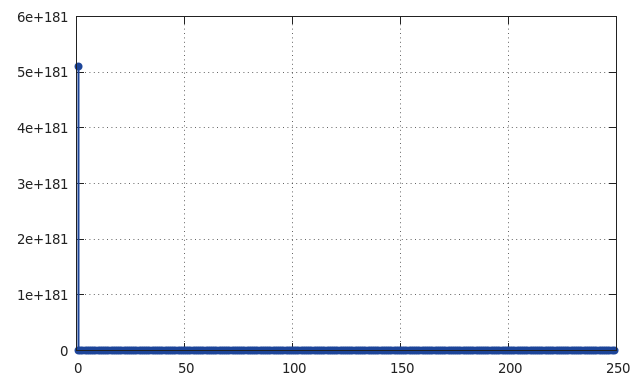


c. Learning Rate = 0.5

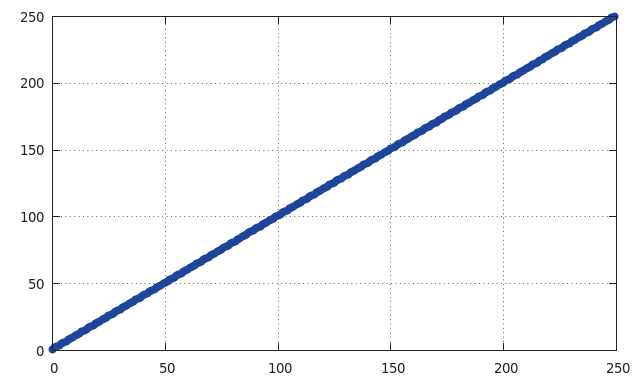


d. Learning Rate = 0.1

e. Learning Rate = 1



f. Learning Rate = 10



**==> Learning Rate = ~1 is the best.**

# Q7. XOR\_Optim

1. Add one more layer (softmax)

inputs = 2; **outputs = 2**; HUs = 20 -- parameters

--1 layer

model:add(nn.Linear(inputs, HUs))

model:add(nn.Tanh())

--2 layer

model:add(nn.Linear(HUs, outputs))

--3 layer

**model:add(nn.Linear(outputs, outputs))**

**model:add(nn.SoftMax())**

2. Adjust Target, Learning Rate and Iterations

**batchSize = 200**

batchInputs = torch.DoubleTensor(batchSize, inputs)

batchLabels = torch.DoubleTensor(batchSize, outputs)

for i = 1, batchSize do

local input = torch.randn(2) -- normally distributed example in 2d

local label

**if input[1] \* input[2] > 0 then -- calculate label for XOR function**

**label = torch.Tensor({1, 0})**

**else**

**label = torch.Tensor({0, 1})**

**end**

batchInputs[i]:copy(input)

batchLabels[i]:copy(label)

end

….

**for epoch = 1, 10000 do**

**…**

**local optimState = {learningRate = 0.05}**

3. Expected Output vs. Actual Output (Nearly perfect)

x = torch.Tensor(2)

x[1] = 0.5; x[2] = 0.5; print(model:forward(x))

x[1] = 0.5; x[2] = -0.5; print(model:forward(x))

x[1] = -0.5; x[2] = 0.5; print(model:forward(x))

x[1] = -0.5; x[2] = -0.5; print(model:forward(x))

Program starting as '"/opt/zbstudio/bin/linux/x64/lua" -e "io.stdout:setvbuf('no')" "/opt/zbstudio/myprograms/XOR\_optim3.lua"'.

Program 'lua' started in '/opt/zbstudio/myprograms' (pid: 19573).

tput: No value for $TERM and no -T specified

**1.0000e+00**

**1.8929e-06**

[torch.DoubleTensor of size 2]

**4.7943e-07**

**1.0000e+00**

[torch.DoubleTensor of size 2]

**3.4422e-07**

**1.0000e+00**

[torch.DoubleTensor of size 2]

**1.0000e+00**

**2.2630e-06**

[torch.DoubleTensor of size 2]

Program completed in 4.87 seconds (pid: 19573).

# Appendix

## Question 5

|  |  |
| --- | --- |
| --Question 5 |  |
|  | --@author @danhtuan |
|  | --@date Feb 7, 17 |
|  | require 'gnuplot' |
|  | require 'torch' |
|  | --gnuplot.closeall() |
|  | --Init |
|  | local R = 1 --#inputs |
|  | local Q = 3 --#data points |
|  | local lr = 0.1 --learning rate |
|  | --w & b |
|  | local x = torch.Tensor(R + 1, 1):fill(0) --homogenious |
|  | --input |
|  | local P = torch.Tensor(Q, 1) |
|  | P[{{}, {1}}] = torch.Tensor({-1, 0, 1}) |
|  | --targets |
|  | local T = torch.Tensor(Q, 1) |
|  | T[{{}, {1}}] = torch.Tensor({-1.5, 0, 2.5}) |
|  |  |
|  | local G = torch.Tensor({{-1, 0, 1}, {1, 1, 1}}):t() |
|  | --A, d, c |
|  | local c = torch.mm(T:t(), T) |
|  | local d = torch.mm(G:t(), T):mul(-2) |
|  | local A = torch.mm(G:t(), G):mul(2) |
|  | --F(x) the loss function (criterion) |
|  | local function floss() |
|  | local F = x:t() \* A \* x |
|  | F = F + torch.mm(d:t(), x) |
|  | F = F + c |
|  | return F |
|  | end |
|  |  |
|  | --gradient |
|  | local function df(var) |
|  | local delta = torch.mm(A, x) |
|  | delta = delta + d |
|  | return delta |
|  | end |
|  |  |
|  | --update w & b |
|  | local function update(grad) |
|  | x = x - grad:mul(lr) |
|  | end |
|  | --train the network |
|  | local runtest = function() |
|  | local sse = torch.Tensor(10) |
|  | --compute gradient |
|  | local grad = df(x) |
|  | local iter = 1 |
|  | sse[iter] = floss() |
|  | while(torch.norm(grad) >= 0.01) do |
|  | iter = iter + 1 |
|  | --update network |
|  | update(grad) |
|  | sse[iter] = floss() |
|  | end |
|  | gnuplot.figure() |
|  | gnuplot.title('SSE vs. Iteration') |
|  | gnuplot.xlabel('iterations') |
|  | gnuplot.ylabel('SSE') |
|  | gnuplot.plot(sse) |
|  | end |
|  | runtest() |
|  | --test the network |
|  | local AP = torch.range(-1.5, 1.5, 0.1) |
|  | local AT = AP:mul(x[1][1]):add(x[2][1]) |
|  | gnuplot.figure() |
|  | gnuplot.title('Network responses vs. targets') |
|  | gnuplot.plot({AP, AT, '-'}, {P:select(2,1), T:select(2,1), '+'}) |

## Question 7

|  |  |
| --- | --- |
| require 'nn' |  |
|  |  |
|  | model = nn.Sequential() -- make a multi-layer perceptron |
|  | inputs = 2; outputs = 2; HUs = 20 -- parameters |
|  | --1 layer |
|  | model:add(nn.Linear(inputs, HUs)) |
|  | model:add(nn.Tanh()) |
|  | --2 layer |
|  | model:add(nn.Linear(HUs, outputs)) |
|  | --3 layer |
|  | model:add(nn.Linear(outputs, outputs)) |
|  | model:add(nn.SoftMax()) |
|  |  |
|  | criterion = nn.MSECriterion() |
|  |  |
|  | batchSize = 200 |
|  | batchInputs = torch.DoubleTensor(batchSize, inputs) |
|  | batchLabels = torch.DoubleTensor(batchSize, outputs) |
|  | for i = 1, batchSize do |
|  | local input = torch.randn(2) -- normally distributed example in 2d |
|  | local label |
|  | if input[1] \* input[2] > 0 then -- calculate label for XOR function |
|  | label = torch.Tensor({1, 0}) |
|  | else |
|  | label = torch.Tensor({0, 1}) |
|  | end |
|  | batchInputs[i]:copy(input) |
|  | batchLabels[i]:copy(label) |
|  | end |
|  |  |
|  | -- Put parameters into vector |
|  | params, gradParams = model:getParameters() |
|  |  |
|  | local optimState = {learningRate = 0.05} |
|  |  |
|  | require 'optim' |
|  |  |
|  | for epoch = 1, 10000 do |
|  | -- local function we give to optim |
|  | -- it takes current weights as input, and outputs the loss |
|  | -- and the gradient of the loss with respect to the weights |
|  | -- gradParams is calculated implicitly by calling 'backward', |
|  | -- because the model's weight and bias gradient tensors |
|  | -- are simply views onto gradParams |
|  | function feval(params) |
|  | gradParams:zero() |
|  |  |
|  | local outputs = model:forward(batchInputs) |
|  | local loss = criterion:forward(outputs, batchLabels) |
|  | local dloss\_doutputs = criterion:backward(outputs, batchLabels) |
|  | model:backward(batchInputs, dloss\_doutputs) |
|  |  |
|  | return loss, gradParams |
|  | end |
|  | optim.sgd(feval, params, optimState) |
|  | end |
|  |  |
|  | x = torch.Tensor(2) |
|  | x[1] = 0.5; x[2] = 0.5; print(model:forward(x)) |
|  | x[1] = 0.5; x[2] = -0.5; print(model:forward(x)) |
|  | x[1] = -0.5; x[2] = 0.5; print(model:forward(x)) |
|  | x[1] = -0.5; x[2] = -0.5; print(model:forward(x)) |